

# Instant Preview and the T<sub>E</sub>X daemon

Jonathan Fine

## Instant Preview

T<sub>E</sub>X is traditionally thought of as a batch program that converts text files into typeset pages. This article describes an add-on for T<sub>E</sub>X, that in favourable circumstances can compile a file in a twentieth of the normal time. This allows T<sub>E</sub>X to be used in interactive programs. This section describes Instant Preview.<sup>1</sup>

**Types of users** Almost all users of T<sub>E</sub>X are familiar with the edit-compile-preview cycle that is part of the customary way of using T<sub>E</sub>X. Previewing is very useful. It helps avoid wasting paper, and it saves time. In the early days, it could take several seconds to compile and preview a file, and perhaps minutes to print it. Today it takes perhaps about a quarter of a second to compile and preview a file.

Many of today's newcomers to computing, and most users of WYSIWYG word processors, expect to have instant feedback, when they are editing a document. Users of T<sub>E</sub>X expect the same instant feedback, when they are editing a source file in a text editor. Because they have absorbed the meaning of the markup codes, they can usually imagine without difficulty the printed form of the document. They know when the markup is right.

Beginners tend to compile the document frequently, because they are uncertain, and wish to have the positive reinforcement of success. Instant Preview, again under favourable circumstances, can reduce to a twentieth the time take to compile and preview a file. This makes it practical to offer preview after every keystroke. Beginners will be able to see their failures and successes as they happen.

Experienced users do not need such a high level of feedback, and prefer to devote the whole screen to the document being edited. However, even experts have the same need for positive reinforcement, when they use a package that is new to them.

**Modus operandi** Here we describe three possible ways of using Instant Preview. At the time of writ-

ing, only the last has been implemented. We assume that the document is in the editing stage of its life cycle, or in other words the location of page breaks and the like is not of interest.

The expert needs only occasionally to preview the source document. She will select the region of interest, and ask for it to be previewed. Instant Preview here may provide a quick and convenient interface, but the operation is uncommon and so the functionality should be unobtrusive.

When doing something tricky, the user might wish to focus on a part of the document, and for this part have Instant Preview after every keystroke. The tuning of math spacing in a formula is an example. Few if any users invariably know, without looking, what tuning should be applied to a moderately complicated formula. This applies particularly to displayed equations wider than the measure, multi-line equations, and commutative diagrams. It also applies to the picture environment (for which the special tool T<sub>E</sub>Xcad was written).

For the beginner, everything is tricky, even straight text. The beginner hardly knows that `\{ } # ^ _ %` are all special characters, and that `“` and `”` are the way to get open and close double quotes. Even experts, who know full well the rules for spaces after control sequences, sometimes make a mistake<sup>2</sup>. The absolute beginner is likely to want Instant Preview of everything, absolutely all the time. Later, with experience, the training wheels can be removed.

**Implementation** Instant Preview has been implemented using Emacs and `xdvi`. There seems to be no reason why another editor and previewer should not be used, provided the editor is sufficiently programmable, and the previewer can be told to refresh the file it is previewing.

Instant Preview works by writing the region to be previewed, together with suitable preamble and postamble, to a special place. From there, the T<sub>E</sub>X daemon picks it up, typesets it, and writes it out as

<sup>1</sup> A screen-shot, in PNG format, is available at the author's website. [It would be nice if it could be included, but I don't know how to do this.]

<sup>2</sup> In the first draft, the allegedly expert author forgot that `&` is also special, and also that `\verb` cannot be used in a L<sup>A</sup>T<sub>E</sub>X footnote.

a dvi file. Once written, the previewer is told to refresh its view of the dvi file.

The main difference between the three modes is what is written out, and when. Absolute beginner mode writes out the whole buffer, after every keystroke. Confident expert mode writes out a selected region, but only on demand.

At the time of writing (mid-June 2001), only absolute beginner mode has been implemented. Further progress requires above all clear goals and Emacs programming skills.

### The dvichop program

For interactive programs, speed is of the essence. Therefore, we will look at T<sub>E</sub>X's performance. The author's computer has a 225MHz Cyrix CPU. So that we have a definite figure, we will say that on this machine a response time of 1/10 seconds is acceptable.

**Typesetting story.tex** There is a file, `story.tex`, that is part of every T<sub>E</sub>X distribution. It is described in *The T<sub>E</sub>Xbook*. On the author's computer, the command

```
time tex ./story \end
```

takes .245 seconds to execute<sup>3</sup>. This seems to make Instant Preview impossible.

However, the command

```
time tex \end
```

takes only .240 seconds to execute. Therefore, it takes T<sub>E</sub>X almost 1/4 of a second to load and exit, while typesetting the two short paragraphs in `story.tex` can be done about 20 times in the target time of a tenth of a second.

Thus, provided the overhead of loading (and exiting) T<sub>E</sub>X can be avoided, Instant Preview is possible.

**Remarks on performance** The simple tests earlier in this article show that it takes T<sub>E</sub>X about 0.005 seconds to typeset the file `story.tex`. This subsection gives a more precise result. It also shows some of the factors that can influence apparent performance.

The file `100story.tex` is as below.

```
\def\0{\input ./story }
\def\1{\0\0\0\0\0\0\0\0\0\0}
\def\2{\1\1\1\1\1\1\1\1\1\1}
\2 \end
```

Table 1 gives the time taken to process this file, in the various modes. It shows that on the au-

<sup>3</sup> To avoid the overhead of X-windows, this command was executed in a virtual console. The same goes for the other timing data. The input file is placed in the current directory to reduce kpathsea overheads.

Mode	seconds
Console, output to /dev/null	.492
Console, output to screen	.507
X-windows, output to /dev/null	.497
X-windows, output to screen	.837

Table 1: Time taken to typeset `story.tex` 100 times

thor's machine and in the best conditions, it takes about  $0.0025 \approx (0.492 - 0.240)/100$  seconds to process `story.tex` once.

Note that the time taken can be quite sensitive to the mode, particularly X-windows. We also note that using `\input story` (so that kpathsea looks for the file) adds about 0.025 seconds to the total time taken.

**Starting T<sub>E</sub>X once** The solution is to start T<sub>E</sub>X once, and use it to typeset multiple documents. Once T<sub>E</sub>X has typeset a page, it uses the `\shipout` command to write it to the dvi file. The new page now exists on the file system, and can be used by other programs. Actually, this is not always true. To improve performance, the system dependent part of T<sub>E</sub>X usually buffers the output dvi stream. However, this can be turned off. We assume that dvi output is unbuffered.

Most dvi-reading applications are unable to process such an ill-formed dvi file. For example, most immediately seek to the end of the file, to obtain a list of fonts used. To bridge this gap, and thereby enable Instant Preview, the author wrote a utility program called `dvichop`.

This program takes as input a dvi file, perhaps of thousands of pages, and produces from it perhaps thousands of tiny dvi files. The little files are the ones that the previewer is asked to reload.

More exactly, `dvichop` looks for special *marker pages* in the output dvi-stream produced by T<sub>E</sub>X the program. The marker pages delimit the material that is to be written to the small dvi files. The marker pages also control where the output of `dvichop` is to be written, and which process is to be informed once the output page is ready.

**Implementation** The program `dvichop` is written in the C programming language. It occupies about 800 lines of code, and calls in a header file `dviop.h` to define the opcodes. A shell program `texd` starts T<sub>E</sub>X and sends its dvi output to `dvichop`. More exactly, T<sub>E</sub>X writes to a named pipe (a FIFO), which is then read by `dvichop`.

**More on performance** In the abstract it is claimed that T<sub>E</sub>X together with dvichop is over 20 times quicker than ordinary T<sub>E</sub>X, when applied to small files. Here is some test data to support this bold claim.

Normally, dvichop is run using a pipe. To simplify matters, we will create the input stream as an ordinary file. The plain input file listed below does this. It also illustrates the interface to dvichop.

```
% 100chop.tex
\newcount\dvicount
\def\0{
\begingroup % begin chop marker page
\global\advance\dvicount 1
\count0\maxdimen \count1 3
\count2 \dvicount \shipout\hbox{ }
\endgroup
\input ./story % typeset the story
\begingroup % end chop marker page
\count0\maxdimen \count1 4
\count2 0 \shipout\hbox{ }
\endgroup
}
\def\1{\0\0\0\0\0\0\0\0\0\0}
\def\2{\1\1\1\1\1\1\1\1\1\1}
\begingroup % say hello to dvichop
\count0\maxdimen \count1 1
\count2 1 \shipout\hbox{ }
\endgroup
\2 % ask dvichop to produce 100 files
\begingroup % say goodbye to dvichop
\count0\maxdimen \count1 2
\count2 0 \shipout\hbox{ }
\endgroup
\end
```

Typesetting `story.tex` 100 times in the conventional way takes approximately 24.5 seconds. Running T<sub>E</sub>X on `100chop.tex` takes about 0.510 seconds. This typesets the story for us 100 times. Running dvichop on the output file `100chop.dvi` takes 0.135 seconds. Its execution creates files `1.dvi` through to `100.dvi` that are for practical purposes identical to those obtained in the conventional way. The conventional route takes 24.5 seconds. The dvichop route took  $0.510 + 0.135 = 0.645$  seconds.

This indicates that on `story.tex` using dvichop is  $24.5/0.635 \approx 38$  times quicker. Some qualifying remarks are in order. In practice, using the pipeline will add overhead, but this seems to be less than 0.01 seconds. On the other hand, the present version of dvichop is not optimised.

## The T<sub>E</sub>X daemon

At this point we assume the reader has some basic familiarity with client-server architecture. A server is a program that is running more or less continually, waiting for requests from clients. Clients can come and go, but servers are expected to persist. An operating system is a classic example of a server, while an application is a client.

**Thanks for the memory** Normally, T<sub>E</sub>X is run as an application or client program. It is loaded into memory to do its job, it does its job, and then it exits. In the mid-1980s, when the author started using a personal computer, having more than a megabyte of memory was uncommon. T<sub>E</sub>X is uncomfortable on less than 512Kb of memory. Thus running T<sub>E</sub>X as a server would consume perhaps half of the available memory. For all but the most rabid T<sub>E</sub>X-ophile, this is clearly not an option.

Today T<sub>E</sub>X requires perhaps 2Mb of memory, and personal computers typically have at least 32Mb of memory. Letting T<sub>E</sub>X remain in memory on a more or less permanent basis, much as Emacs and other programs remain loaded even when not used, is clearly practical. However, even today, for most users there is probably not room to have more than a handful of instances of T<sub>E</sub>X resident in memory.

**Sockets** The present implementation of Instant Preview uses a named pipe. Sockets provide a more reliable and flexible interface. In particular, sockets can handle contention (requests to the same server from several clients). Applications communicate to the X-server provided by X-windows by means of a socket.

Providing a socket interface to the T<sub>E</sub>X daemon will greatly increase its usefulness. The author hopes that by the end of the year he or someone else will have done this.

**T<sub>E</sub>X as a callable function** Over the years, many people have complained that the batch nature of T<sub>E</sub>X makes it unsuitable for today's new computing world. They have wanted T<sub>E</sub>X to be a callable function. However, to make T<sub>E</sub>X a callable function, all that is required is a suitable wrapper, that communicates with the T<sub>E</sub>X daemon.

At present the T<sub>E</sub>X daemon is capable of returning only a `dvi` file. To do this, it must parse the output `dvi` stream. Suppose, for example, that the caller wants to convert the output `dvi` into a bitmap, say for inclusion in an HTML page. The present set-up would result in the `dvi` pages being

parsed twice. Although this is not expensive, compared to starting up a whole new  $\text{\TeX}$  process, it is still far from optimal.

If the  $\text{\TeX}$  daemon could be made to load page-handling modules, then the calling function could then ask for the bitmap conversion module to handle the pages produced by the function call. This would be more efficient. However, as we shall soon see, premature optimisation can be a source of problems.

**$\text{\TeX}$  forever** An errant application does not bring down the operating system. Strange keystrokes and mouse movements do not freeze X-windows. In the same way, applications should never be able to kill the  $\text{\TeX}$  daemon. To achieve this level of reliability is something of a programming problem.

One thing is clear: The application cannot be allowed to send arbitrary raw  $\text{\TeX}$  to the  $\text{\TeX}$  daemon.  $\text{\TeX}$  is much too sensitive. All it takes is something like

```
\global\let\def\undefined
```

and the  $\text{\TeX}$  daemon will be rendered useless.

A more subtle form of this problem is when a client's call to the daemon results in an unintended, unwelcome, and not readily reversible change of state. For example, the  $\text{\LaTeX}$  macro `\maketitle` executes

```
\global\let\maketitle\relax
```

which is an example of such a command. (Doing this frees tokens from  $\text{\TeX}$ 's main memory. When  $\text{\TeX}$ , macros and all, is shoe-horned into 512Kb, this may be a good idea.)

**Protecting  $\text{\TeX}$**   $\text{\TeX}$  can be made a callable function by providing an interface to the  $\text{\TeX}$  daemon. Most applications will want an interface that is safe to use. In other words, input syntax errors are reported before they get to  $\text{\TeX}$ , and it is not possible to accidentally kill the  $\text{\TeX}$  daemon. To provide this, the interface must be well defined. For example, the input might be an XML-document (say as a string) together with style parameters, and the output would be say a  $\text{\dvi}$  file. Alternatively, the input might be a pointer to an already parsed data structure.

In the long run, this interface is probably best implemented using compiled code, rather than  $\text{\TeX}$  macros. Once a function is used to translate source document into  $\text{\TeX}$  input, there is far less need for developers to write complicated macros whose main purpose is to provide users with a comfortable input syntax. Instead, the interface function can do this.

When carried out in a systematic manner, this will remove the problem, that in general  $\text{\LaTeX}$  is the only program that can understand a  $\text{\LaTeX}$  input

file. The same holds for other  $\text{\TeX}$  macros formats, of course. Note that Don Knuth's **WEAVE** (part of his literate programming system) is similarly compiled code that avoids the need to write complicated  $\text{\TeX}$  macros.

## Visual $\text{\TeX}$

This article uses the term visual  $\text{\TeX}$  to mean programs and other resources that allow the user to interact with a document through a formatted representation, typically a previewed  $\text{\dvi}$  file. We use it in preference to WYSIWYG (what you see is what you get) for two reasons. The first is today many documents are formatted only for screen, and never get printed. Help files and web pages are examples of this. The second is that even when editing a document for print, the user may prefer a representation that is not WYSIWYG.

In most cases the author will benefit from interacting with a suitably formatted view of the underlying document. The benefits of readability and use of space that typesetting provides in print also manifest on the screen. But to insist on WYSIWYG is to ignore the differences between the two media. Hence our use of the term Visual  $\text{\TeX}$ .

Whatever term is used, the technical problems are much the same, which is how to enable user interaction with the  $\text{\dvi}$  file.

**Richer  $\text{\dvi}$  files** In Visual  $\text{\TeX}$ , the resulting  $\text{\dvi}$  file is a view on the underlying document. For it to be possible to edit the document through the view, the view must allow the access to the underlying document. Editing changes applied to the view, such as insertion and deletion, can then be applied to the document.

Placing large numbers of `\special` commands in the  $\text{\dvi}$  file is probably the best (and perhaps the only) way to make this work. Doing this is the responsibility of the macro package (here taken to include the input filter function described in the previous section). It is unlikely that any existing macro package, used in its intended manner, will support the generation of such enriched  $\text{\dvi}$  files. The author's Active  $\text{\TeX}$  macro package[2] is designed to allow this.

**Better  $\text{\dvi}$  previewers** Most  $\text{\dvi}$  previewers convert the  $\text{\dvi}$  into a graphics file, such as a bitmap. Some retain information about the font and position of each glyph. A text editor or word processor has a cursor (called point in Emacs), and by moving the cursor text can be marked. This is a basic property of such programs. So far as the author knows, no  $\text{\dvi}$  previewer allows such marking of text.

**Further reading** This section is based on the author's article [1].

The Lyx editor for L<sup>A</sup>T<sub>E</sub>X adopts a visual approach to the generation of files that can be typeset using L<sup>A</sup>T<sub>E</sub>X. It does not support WYSIWYG interaction. Understanding the capabilities and limitations of Lyx is probably a good way to learn more about this area.

### The next steps

This section discusses some of the opportunities and problems in this general area, likely to present themselves over the next year or two.

**Applications** Two areas are likely to be the focus of development in the next year or so. The first is the refinement of Instant Preview, as a tool for use with existing T<sub>E</sub>X formats. Part of this is the creation of material for interactive (La)T<sub>E</sub>X training. Instant Preview provides an attractive showcase for the abilities of T<sub>E</sub>X and its various macro packages.

The second is T<sub>E</sub>X as a callable function. This is required for Visual T<sub>E</sub>X. One of the important missing components are libraries that allow rich interaction with dvi files. This will lay the foundation for T<sub>E</sub>X being embedded in desktop applications.

**License** The work described in this article is at present released under the General Public Licence of the Free Software Foundation (the GPL). Roughly speaking, this means that any derived work that contains say the author's implementation of the T<sub>E</sub>X daemon must also be released under the GPL.

However, the T<sub>E</sub>X daemon is the basis for T<sub>E</sub>X as a callable function, and for good reason library functions are usually released under the Lesser (or Library) General Public Licence (the LGPL), or something similar. This means that the library as is can be linked into proprietary programs, but that any enhancement to the library must be released under the LGPL.

**Porting** T<sub>E</sub>X runs on almost all computers, and where it runs, it gives essentially identical results. The same applies, of course, to T<sub>E</sub>X macros. By and large, it is desirable that the tools used with T<sub>E</sub>X run can be made to run identically on all platforms. This is not to say that the special features of any particular platform should be ignored. Nor is it to say that advances (such as Instant Preview itself) should not first manifest on a more suitable platform.

Cross-platform portability is one of the great strengths of T<sub>E</sub>X. What is desirable is that programs that run with T<sub>E</sub>X have a similar portability. Many people cannot freely choose their computing

platform. If T<sub>E</sub>X and friends are available everywhere, this makes T<sub>E</sub>X a more attractive choice.

In the 1980s, in the early days of T<sub>E</sub>X, many pioneers ported T<sub>E</sub>X to diverse platforms. This work established deep roots that even today continue to nourish the community. Although Instant Preview, even when fully developed, is not on the same scale as T<sub>E</sub>X, it being ported will similarly nourish the community.

**T<sub>E</sub>X macros** Visual T<sub>E</sub>X requires a stable T<sub>E</sub>X daemon, which in turn will require a macro package (or a pre-loaded format). This new use of T<sub>E</sub>X places new demands on the macros. Here, we include in macros any input filter functions used to protect the T<sub>E</sub>X daemon from errant applications.

These new demands include protection against change of state, reporting and recovery from errors, ability to typeset document fragments, support for rich dvi file, and the ability for a single daemon to support round-robin processing of multiple documents. Once tools are in place, much of the input is likely to be XML, and much of the output will be for screen rather than paper.

The existing macros packages (such as plain, L<sup>A</sup>T<sub>E</sub>X and ConT<sub>E</sub>XT) were not written with these new requirements in mind. Although they are useful now, in the longer term it may be better to write a new macro package from scratch, for use in conjunction with suitable input filters.

### Summary

By running T<sub>E</sub>X within a client-server architecture, many of the problems traditionally associated with it are removed. At the same time, new demands are placed on macro packages, device drivers (such as dvichop and xdvi) and a new category of software, input filters (such as WEAVE).

This new architecture allows Instant Preview, and opens the door to Visual T<sub>E</sub>X. All this is possible without making any changes to T<sub>E</sub>X the program, other than in the system dependent part.

**Don Knuth** In 1990, when he told us [4] that his work on developing T<sub>E</sub>X had come to an end, Don Knuth went on to say:

Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

The work described in this article has taken its direction from this statement. One of the most obvious characteristics of today's computer monitors (not to be confused with the chalk monitor in classrooms of old) is their widespread use of colour. T<sub>E</sub>X is clumsy with colour. T<sub>E</sub>X was not designed with Visual T<sub>E</sub>X in mind. However, we still have our hands full making the best of what we have with T<sub>E</sub>X. If our labours bear fruit, then in time a place and a need for a successor will arise.

Again, this possibility was foretold by Don Knuth [3]:

Of course I don't mean to imply that all problems of computational typography have been solved. Far from it! There are still countless important issues to be studied, relating especially to the many classes of documents that go far beyond what I ever intended T<sub>E</sub>X to handle.

## References

- [1] Jonathan Fine, Editing .dvi files, or Visual T<sub>E</sub>X, *TUGboat*, **17** (3) (1996), 255–259.
- [2] ———, Active T<sub>E</sub>X and the D<sub>OT</sub> input syntax, *TUGboat*, **20** (3) (1999), 248–261
- [3] Donald E. Knuth, The Errors of T<sub>E</sub>X, *Software—Practice & Experience*, **19** (1989) 607–685 (reprinted in *Literate Programming*)
- [4] ———, The future of T<sub>E</sub>X and METAFONT, *TUGboat*, **11** (4) (1990), 489 (reprinted in *Digital Typography*)